An Approach for Data-driven and Logic-based Complex Event Processing

Darko Anicic[1]
darko.anicic@fzi.de

Paul Fodor[2]
pfodor@cs.sunysb.edu

Roland Stühmer[1]
roland.stuehmer@fzi.de

Nenad Stojanovic[1]
nenad.stojanovic@fzi.de

[1]FZI Forschungszentrum Informatik, Haid-und-Neu-Straße 10-14, 76131 Karlsruhe, Germany
[2]Department of Computer Science, Stony Brook University, Stony Brook, NY 11794-4400, U.S.A

ABSTRACT

In this paper, we present a powerful logic-based approach for Complex Event Processing (CEP). The approach is founded on a logical transformation of event processing into logic programs. Many systems for event processing have ad-hoc semantics with unexpected behaviors. Further on, many formal approaches (with well defined semantics) cannot effectively be used in CEP, due to their inability to compute events in the data-driven fashion. Our approach enables both logic-based and data-driven complex event detection. Moreover, the backward chaining approach allows for very efficient reasoning of complex events and actions triggered by these events.

1. INTRODUCTION

Complex Event Processing (CEP) has the task of processing multiple events with the goal of identifying meaningful event patterns. Detected event patterns are then used to trigger response actions, which in turn may trigger other events. In this cycle, a reactive system is supposed to achieve some useful job. It has been recognized elsewhere [1, 3] that some sort of *logic* is required to keep event-driven systems running in a controlled manner. The logic should ensure *correctness* of an event-driven execution. It means that the execution is handled in a way which guarantees satisfiability of predefined constraints on states (in a state-changing environment). A number of declarative approaches [3, 1] utilize different kinds of logic to archive afore mentioned goals. However declarative approaches also have their own limitations. The main common drawback is their inability to do an effective *event or data-driven* computation of event patterns. Data-driven computation means that an event pattern is detected as soon as the last event required for a complete match of that pattern has occurred. Logic-based approaches attempt to *deduce* an event pattern rater than to *detect* it. This is why they have typically been suitable for *request-response* systems. These systems detect an event (or situation) based on a request, followed by an action which is a response. However, they fail when patterns need to be detected as soon as they really occur.

In this paper we present an approach which is based on a *logic*, and still it is *event or data-driven*. The approach is based on decomposition of complex event patterns into *intermediate patterns* (i.e., *goals*). The status of achieved goals at the current state shows the progress toward matching of one or more event patterns. Goals are automatically asserted as relevant events occur. They can persist over a period of time "waiting" in order to support detection of a more complex goal or complete pattern. Important characteristics of these goals are that they are asserted only if they are used later on (to support a more complex goal or an event pattern), goals are all unique, and goals persist as long as they remain relevant (after the relevant period they are deleted). Goals are asserted by declarative rules, which are executed in the backward chaining mode. A specific property of these rules is that they are *event-driven*.

DEBS '09 Nashville, TN, USA

2. LOGIC-BASED EVENT-DRIVEN CEP

The execution in CEP applications is driven by events. In order to react on an event, we need first to specify event patterns which we want our application to react on. An *event pattern* is a template which matches certain events. For example, an event pattern matches all orders from customers in response to a discount announcement event. Sometimes the discount announcement event is called an *atomic* event, which is used to build a *complex* event. In general, an event pattern is a pattern which is built out of (atomic and complex) events satisfying certain relational operators, contexts, and data parameters. Currently we support the following set of event operators (the list, in this abstract, is not complete)[1]: $a \otimes b$ (sequence of events), $a \wedge b$ (classical conjunction of events), $a|b$ (concurrent conjunction of events), $a \vee b$ (disjunction of events), and $\neg a$ (negation). A rule is a formula of the form $a \leftarrow b$, where a is an atomic event, and b is either an atomic or a complex event pattern. Every event a, b is defined over a time interval $[T_1, T_2]$ with possible set of data terms, that are omitted due to space reasons. Event patterns contain data relevant for a reactive system[2]. The data of an event pattern is a data term that may be either a variable, a constant, or a function symbol.

In the following, we present our approach, based on a *goal-directed event-driven* rules. The approach is established on decomposition of complex event patterns into *two-input intermediate events* (i.e., *goals*). The status of achieved goals at the current state shows the progress toward completeness of an event pattern. Goals are automatically asserted by rules as relevant events occur.

[1]For a complete list of operators and implementation, see our full technical report: http://code.google.com/p/etalis/wiki/TechnicalReport?ts=1241893834&updated=TechnicalReport.

[2]We avoid representing data due to space reasons.

They can persist over a period of time "waiting" in order to support detection of a more complex goal or pattern.

Let us consider our example rules (1). e_1 represents a conjunction of e_2 and e_3, while the conjunct further represent two sequences of a and b, and further of c and d. In their present form, these rules are not convenient to be used for *event-driven* computation. These are rather a Prolog-style rules suitable for backward chaining evaluation. Such rules are understood as goals which at certain time either can or cannot be proved by an inference engine. The difficulty is that such an inference process cannot be done in an *event-driven* fashion.

$$\begin{array}{l} e1([T_5,T_6]) \leftarrow e2([T_1,T_2]) \wedge e3([T_3,T_4]). \\ e2([T_1,T_4]) \leftarrow a([T_1,T_2]) \otimes b([T_3,T_4]). \\ e3([T_1,T_4]) \leftarrow c([T_1,T_2]) \otimes d([T_3,T_4]). \end{array} \quad (1)$$

Rules (2)-(9) represent our example rules (1), rewritten so to enable *event-driven backward chaining*. First, they are suitable for detection of events as soon as they occur. Second, detection of events is focused on patterns of interest, rather than merely detecting irrelevant events. Each rule from (2)-(9) falls in one of two types of rules. One type is used to generate goals, and another one generates either intermediate events (e.g., $e2$, $e3$) or pattern events (e.g., $e1$).

The first type of rules are used to generate goals (i.e., insert goals into database). Every generated goal represents that an event has occurred. For example, rule (2) inserts $goal(b([\_,\_]), a([T_1,T_2]), e2([\_,\_])))$ every time an event of type a occurs. A goal consists of three events (terms). Its interpretation is that "an instance of event a has occurred at $[T_1,T_2]$, and we are waiting for b to happen in order to detect $e2$". In general, the second event in a goal always denotes an event that has just occurred. The role of the first event is to specify *what* we are waiting for to detect an event that is on the third position.

The second type of rules checks whether a certain goal already exists in the database, in which case it triggers an event. For example, the second rule in rules (3) will fire whenever b occurs. The rule checks whether $goal(b([\_,\_]), a([T_1,T_2]), e2([\_,\_]))$ already exists (i.e., a previously has happened), in which case it triggers $e2$ (by calling $e2([T_1,T_4])$. The time occurrence of $e2$ (i.e., $[T_1,T_4]$) is defined based on the occurrence of constituting events (i.e., $a([T_1,T_2])$, and $b([T_3,T_4])$). Calling $e2([T_1,T_4])$, this event is effectively propagated either upward (if it is an intermediate event) or used for triggering an action (if it is an event pattern, e.g., $e1$).

While rules (2)-(5) are used to detect the two sequences, rules (7) and (8) detect conjunction from our example. Let us assume that the sequence $e2$ was detected as the first one (followed by $e3$ afterwards). In this situation, rules (7) will fire. The loop will execute two rules. The first rule will check whether $e3$ has previously happened, in which case it will trigger $e1([T_5,T_6])$, provided that $T_5 = min(T_1,T_3)$ and $T_5 = max(T_2,T_4)$. "Consumed" goals are set to be removed from the database once they become irrelevant (if necessary, this can be avoided by omitting del operations). However if $e3$ had not previously happened (by the time when $e2$ occurred), the second rule will insert $goal(e3([\_,\_]), e2([T_3,T_4]), e1([\_,\_]))$. This goal has the meaning that $e2$ occurred in $[T_3,T_4]$, and now we are waiting for $e3$ to happen in order to detect $e1$. Once $e3$ occurs, rules (8) will execute. Again there is a loop firing two rules. The first rule will trigger $e1$, provided that $e2$ has already happened. A time interval on which $e1$ has been detected is again $T_5 = min(T_1,T_3)$ and $T_6 = max(T_2,T_4)$. If $e2$ has not occurred, a goal stating that $e3$ happened (in scope of detection of $e1$) is inserted[3].

$$\begin{array}{l} a([T_1,T_2]) : -while\_do(a,1,[T_1,T_2]). \\ a(1,[T_1,T_2]) : -ins(goal(b([\_,\_]), a([T_1,T_2]), e2([\_,\_]))). \end{array} \quad (2)$$

$$\begin{array}{l} b([T_1,T_2]) : - while\_do(b,1,[T_1,T_2]). \\ b(1,[T_3,T_4]) : - goal(b([\_,\_]), a([T_1,T_2]), e2([\_,\_])) \otimes \\ del(goal(b([\_,\_]), a([T_1,T_2]), e2([\_,\_]))) \otimes e2([T_1,T_4]). \end{array} \quad (3)$$

$$\begin{array}{l} c([T_1,T_2]) : - while\_do(c,1,[T_1,T_2]). \\ c(1,[T_1,T_2]) : - ins(goal(d([\_,\_]), c([T_1,T_2]), e3([\_,\_]))). \end{array} \quad (4)$$

$$\begin{array}{l} d([T_1,T_2]) : - while\_do(d,1,[T_1,T_2]). \\ d(1,[T_3,T_4]) : - goal(d([\_,\_]), c([T_1,T_2]), e3([\_,\_])) \otimes \\ del(goal(d([\_,\_]), c([T_1,T_2]), e3([\_,\_]))) \otimes e3([T_1,T_4]). \end{array} \quad (5)$$

$$\begin{array}{l} e1([T_1,T_2]) : -while\_do(e1,1,[T_1,T_2]). \\ e1(1,[T_1,T_2]) : -action1([T_1,T_2]). \end{array} \quad (6)$$

$$\begin{array}{l} e2([T_1,T_2]) : -while\_do(e2,1,[T_1,T_2]). \\ e2(1,[T_3,T_4]) : -goal(e2([\_,\_]), e3([T_1,T_2]), e1([\_,\_])) \otimes \\ del(goal(e2([\_,\_]), e3([T_1,T_2]), e1([\_,\_]))) \otimes min(T_1,T_3,T_5) \otimes \\ max(T_2,T_4,T_6) \otimes e1([T_5,T_6]). \\ e2(1,[T_3,T_4]) : -not(goal(e2([\_,\_]), e3([T_1,T_2]), e1([\_,\_]))) \otimes \\ ins(goal(e3([\_,\_]), e2([T_3,T_4]), e1([\_,\_]))). \end{array} \quad (7)$$

$$\begin{array}{l} e3([T_1,T_2]) : -while\_do(e3,1,[T_1,T_2]). \\ e3(1,[T_3,T_4]) : -goal(e3([\_,\_]), e2([T_1,T_2]), e1([\_,\_])) \otimes \\ del(goal(e3([\_,\_]), e2([T_1,T_2]), e1([\_,\_]))) \otimes min(T_1,T_3,T_5) \\ max(T_2,T_4,T_6) \otimes e1([T_5,T_6]). \\ e3(1,[T_3,T_4]) : -not(goal(e3([\_,\_]), e2([T_1,T_2]), e1([\_,\_]))) \otimes \\ ins(goal(e2([\_,\_]), e3([T_3,T_4]), e1([\_,\_]))). \end{array} \quad (8)$$

$$\begin{array}{l} min(T_1,T_2,T_3) : -(T_1 < T_2 \rightarrow T_3 = T_1; T_3 = T_2). \\ max(T_1,T_2,T_3) : -(T_1 > T_2 \rightarrow T_3 = T_1; T_3 = T_2). \end{array} \quad (9)$$

Transformed rules (2)-(5) are, in practice, automatically generated for a set of user defined rules (e.g., rules (1)). In the mentioned technical report we have described an algorithm for each event operator of our logic-based event processing framework. Further on, we have provided an implementation of our approach, and shown the first evaluation results. The implementation is based on Prolog and Concurrent Transaction Logic ($\mathcal{CTR}$) [2], that is a general logic designed specifically for the rule-based paradigm. First results are more than encouraging, and we aim to continue on this work. Particularly we will provide definitions for new event operators as well as an optimization of the current implementation.

3. ACKNOWLEDGMENT

We thank Ahmed Khalil Hafsi and Jia Ding for their help in the implementation and testing of the prototype.

4. REFERENCES

[1] E. Behrends, O. Fritzen, W. May, and F. Schenk. Combining eca rules with process algebras for the semantic web. In *RuleML*, 2006.

[2] A. J. Bonner and M. Kifer. Concurrency and communication in transaction logic. In *Joint International Conference and Symposium on Logic Programming*. MIT Press, 1996.

[3] P. Haley. Data-driven backward chaining. In *International Joint Conferences on Artificial Intelligence*. Milan, Italy, 1987.

[3]It is worth noting that a non-occurrence of an event is checked by using a negated goal.